CS 122 – Lab #9 – Java Classes

This exercise will give you practice with classes and aggregation.

Java is an object-oriented programming language. Object orientation means that we solve problems by first looking for nouns before looking for verbs. In object-oriented programming, we create classes to represent our own custom data types. A class consists of various attributes and operations, and individual members of a class are called objects.

The attributes of a class may be of a primitive type, or they may be objects themselves. When objects contain other objects, this class relationship is called *aggregation* (not to be confused with another class relationship called inheritance). For example, your program might define a Hotel class. One of the attributes of this class would be an array of Rooms. And the Room objects themselves have their own attributes.

Here are the steps we follow to create a class. Note that some of these features are optional, depending on the application.

1. Declare the attributes. But do not assign values to these attributes yet. In particular, if an attribute is a data type that we would use the Java keyword `new` to allocate space for, the call to `new` (and any constructor call) would wait until we implement our constructor.
2. Write the constructor(s): default, initial-value, copy. Of these three constructor types, the initial-value constructor is the most common. A constructor is where we allocate space and assign values to all the attributes. Even if you don't know what value to set an attribute to, it's always a good idea to set it to something like zero or null, rather than forget to initialize it. For example, let's say you just hired an employee. Some attributes about the person you already know, such as name. But an office location might not initially be assigned on the day the person is hired, so it could initially be the empty string. This is why we might like to have "set" methods, to reset an attribute value.
3. Get and set methods (also called accessor and mutator methods, respectively).
4. Implement toString(). This often entails the use of the built-in method String.format(). If we have a lot of strings to concatenate, such as in a loop, we should employ the StringBuilder class for sake of efficiency.
5. Implement equals(). Note that we don't need to implement a general compare method, because this is often handled by a separate comparator class. Comparators are beyond the scope of this exercise.

After you have implemented your class, you are free to declare objects of this type elsewhere in your program. In particular, you might be interested in having an ArrayList of your objects. For example, if you have implemented a Hotel class, you might want to have several hotels like this:

```
ArrayList<Hotel> hList = new ArrayList<Hotel>();
```

Note that in this statement we are calling the ArrayList constructor, not the Hotel constructor. After this statement you could write a loop in which the individual Hotel objects are created using a constructor, and then inserted into the list.

Let's create a simple application that illustrates class creation and some aggregation. The file players.txt contains a list of 15 basketball players. Each player has a number, a first name and a last name. We will create three teams of 5 players each. The program will consist of 3 source files: Player.java, Team.java and Driver.java. In other words, we will define a Player class, a Team class, and finally a Driver class that will contain our main program.

Here is what you need to do to create the Player class. Please don't hesitate to ask for help if you need help accomplishing these steps.

1. Declare three attributes: firstName, lastName and number. What types do you think they should be? Be sure to declare these attributes as private.

2. Just for fun, let's make a default constructor. It will not actually be used in our program, but this will be good practice all the same. The default constructor takes no parameters. In this case, it will just set the attributes for a player named John Doe whose number is 1.

3. Most importantly, we need to create an initial-value constructor. Its parameters will have exactly the same names as the attributes: firstName, lastName and number. Initialize each attribute to be equal to the corresponding parameter. You may wonder how the compiler will be able to distinguish between firstName being the name of a parameter into this method and firstName being one of the attributes of the class. The way we will distinguish is by using the Java keyword `this`. When you say `this.firstName`, there is no mistake: you mean the class attribute firstName. And when you simply say firstName, this is the parameter. In general, when we redeclare an identifier in this manner, the local name overrides the global name, hence the need to use "`this`". We could have decided to use different names for attributes and parameters, but this can be error prone if you write the assignment statements backwards! That would be a difficult bug to find.

4. Let's now write a third constructor. It is called a copy constructor, because it allows us to create an object identical to an existing one. This is not as commonly used as the initial-value constructor, but it turns out that we will need it in this program. This constructor will take exactly 1 parameter: a Player object. The attributes of "this" will each be initialized to the corresponding attributes of the parameter object.

5. The next useful thing to add to our Player class is a set of "get" methods. This will be a way for code outside this class to access the values of attributes, since they were declared to be private.

6. If we wanted to be thorough in our class design, we should also provide "set" methods as well. However, they will not be useful in our program, and you may omit these.

7. It's important to be able to provide a text representation of a Player object. We do so by implementing a method called toString(). Since a Player consists of just three simple attributes, the body of toString() can be accomplished in just one statement. We just need to return a string containing all of the attribute values. In order to make the output look attractive, let's agree to the following output format. The player's number should be in parentheses. We should prepare for up to a 2-digit number. Scanning the list of names, we see that they may be up to 12 characters. So, tell Java you want to allocate 12 characters for each name. An example string that we would return is "(12) Schneiderman, Clara ". Use the built-in String.format() method to specify the format.

8. Finally, let's write an equals() method. This takes a Player object as a parameter, which we can call p, for example. The objects this and p are equal if all of their corresponding attributes are equal.

Question: In step 7, you implemented the toString() method of the Player class. Often, this string being returned will end in several space characters. Suppose the calling environment did not want all of those extra spaces at the end of the player's name. What is a simple way to get rid of them after we call toString()?

Nest, let's implement the Team class.

1. What are the attributes of a team? It has a name, and a group of players. What do we mean by a "group"? We could use any data structure, but for simplicity let's just use an array of Player objects. So, I recommend you create two attributes: one called name, and one called playerList.

2. We need a constructor for the Team class. As usual, it will be an initial value constructor. It will take two parameters, corresponding to our two class attributes. The parameters will be a name and an array of Player objects.
   a. Set the name attribute equal to the name parameter.
   b. Allocate space for the attribute playerList. It should be based on the size (i.e. length) of the parameter array.
   c. Write a loop that places each Player object in the parameter array into the attribute array. Be sure to invoke the Player copy constructor each time.

3. Write get methods for the name and the player list.

4. Lastly, let's write a toString() method. What is special here is that you will practice how to use the StringBuilder class.

a. Since a Team consists of several Player objects, we will have a lot of strings to concatenate. For efficiency, let's use StringBuilder. This is a class that is automatically included in java.lang, so you do not need to explicitly import this at the top of the source file. Create a new StringBuilder object called build, using its default constructor. We will begin with this empty StringBuilder object, and then "append" additional strings to it one by one.
b. If you look at the API documentation on the StringBuilder class, you will notice a lot of instance methods called append. We will use the one that takes a String parameter. The first thing we will append to build is the team name and a newline character.
c. Set up a loop that does the following.
   For each element in the this.playersList array,
      call build.append() and pass it playersList[i].toString() and a newline character.
d. Return build.toString().

Question: In step 2(c) above, we used the Player copy constructor. How could we have used the Player initial-value constructor instead? Why might using the copy constructor be the better choice?

Question: Why did we need to call playersList[i].toString() in step 4(c) above?

The last source file we need to write is our Driver file, containing main(). Our program is going to be rather ad hoc, since we know exactly how the input looks. We have 15 players, and we want to create 3 teams of 5. In more general situations, we would probably use an ArrayList in place of an array.

1. Create an array called teamList containing enough space for 3 Team objects. Also, create an array called playerList containing enough space for 5 Player objects.

2. Set up a nested loop as follows.
   for i = 0 to 2, inclusive
      read the next line of input, and store this in a variable called teamName.
      for j = 0 to 4, inclusive
         let line = the next line of input
         Tokenize the line to grab the player's number, first name and last name
         Create a new Player object p from this information
         initialize playerList[j] equal to p
      Create a Team object t, using the initial-value constructor, passing it the name and playerList
      set teamList[i] equal to t

3. Print out the contents of the teamList. For each Team object, call its toString() method.

That's it! Compile and run the program. In your opinion, what was the most difficult task that you had to perform?

One more thing! In the long run, it's more likely that we will collect objects into an ArrayList rather than an array. Where in your code would you have to make changes if we wanted to have an ArrayList of teams, and if each team was to be an ArrayList of players?